15. Areas

Storage in the Lisp Machine is divided into *areas*. Each area contains related objects, of any type. Areas are intended to give the user control over the paging behavior of his program, among other things. Putting frequently used data and rarely used data in different areas can cause the frequently used data to occupy fewer pages. For example, the system puts the debugging info alists of compiled functions in a special area so that the other list structure the functions point to will be more compact.

Whenever a new object is created the area to be used can optionally be specified. For example, instead of using **cons** you can use **cons-in-area** (see page 62). Object-creating functions which take keyword arguments generally accept a **:area** argument. You can also control which area is used by binding **default-cons-area** (see page 224); most functions that allocate storage use the value of this variable, by default, to specify the area to use.

There is a default Working Storage area that collects those objects that the user has not chosen to control explicitly.

Areas also give the user a handle to control the garbage collector. Some areas can be declared to be "static", which means that they change slowly and the garbage collector should not attempt to reclaim any space in them. This can eliminate a lot of useless copying. A "static" area can be explicitly garbage-collected at infrequent intervals when it is believed that that might be worthwhile.

Each area can potentially have a different storage discipline, a different paging algorithm, and even a different data representation. The microcode will dispatch on an attribute of the area at the appropriate times. The structure of the machine makes the performance cost of these features negligible; information about areas is stored in extra bits in the memory mapping hardware where it can be quickly dispatched on by the microcode; these dispatches usually have to be done anyway to make the garbage collector work and to implement invisible pointers. This feature is not currently used by the system, except for the list/structure distinction described below.

Each area has a name and a number. The name is a symbol whose value is the number. The number is an index into various internal tables. Normally the name is treated as a special variable, so the number is what is given as an argument to a function that takes an area as an argument. Thus, areas are not Lisp objects; you cannot pass an area itself as an argument to a function; you just pass its number. There is a maximum number of areas (set at cold-load generation time); you can only have that many areas before the various internal tables overflow. Currently (as this manual is written) the limit is **256.** areas, of which **64.** already exist when you start.

The storage of an area consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties. The most important of these is the *data representation type*. A given region can only store one type. The two types that exist now are *list* and *structure*. A list is anything made out of conses (a closure for instance). A structure is anything made out of a block of memory with a header at the front; symbols, strings, arrays, instances, compiled functions, etc. Since lists and structures cannot be stored in the same region, they cannot be on the same page. It is necessary to know about this when using areas to increase

locality of reference.

When you create an area, one region is created initially. When you try to allocate memory to hold an object in some area, the system tries to find a region that has the right data representation type to hold this object, and that has enough room for it to fit. If there isn't any such region, it makes a new one (or signals an error; see the :size option to make-area, below). The size of the new region is an attribute of the area (controllable by the :region-size option to make-area). If regions are too large, memory may get taken up by a region and never used. If regions are too small, the system may run out of regions because regions, like areas, are defined by internal tables that have a fixed size (set at cold-load generation time). Currently (as this manual is written) the limit is 256. regions, of which about 90. already exist when you start. (If you're wondering why the limit on regions isn't higher than the limit on areas, as it clearly ought to be, it's just because both limits have to be multiples of 256. for internal reasons, and 256. regions seem to be enough.)

15.1 Area Functions and Variables

default-cons-area *Variable*

The value of this variable is the number of the area in which objects are created by default. It is initially the number of working-storage-area. Giving nil where an area is required uses the value of default-cons-area. Note that to put objects into an area other than working-storage-area you can either bind this variable or use functions such as cons-in-area (see page 62) which take the area as an explicit argument.

background-cons-area *Variable*

The value of this variable is the number of a non-temporary area in which objects created as incidental side effects by system functions should be created. This area is used whenever an object is created that should never be in a temporary area, even if *default-cons-area* is a temporary area.

By default, this area is working-storage-area.

make-area &rest *keywords*

Creates a new area, whose name and attributes are specified by the keywords. You must specify a symbol as a name; the symbol will be setq'ed to the area-number of the new area, and that number will also be returned, so that you can use make-area as the initialization of a defvar. The arguments are taken in pairs, the first being a keyword and the second a "value" for that keyword. The last three keywords documented herein are in the nature of subprimitives; like the stuff in chapter 14, their meaning is system-dependent and is not documented here. The following keywords exist:

:name — A symbol that will be the name of the area. This item is required.

:size — The maximum allowed size of the area, in words. Defaults to infinite. (Actually, the default is the largest positive fixnum; but the area is not limited to that size!) If the number of words allocated to the area reaches this size, attempting to cons an object in the area will signal an error.

:region-size — The approximate size, in words, for regions within this area. The default is the area size if a :size argument was given, otherwise it is a suitable

medium size. Note that if you specify :size and not :region-size, the area will have exactly one region. When making an area that will be very big, it is desirable to make the region size larger than the default region size to avoid creating very many regions and possibly overflowing the system's fixed-size region tables.

:representation
The type of object to be contained in the area's initial region. The argument to this keyword can be :list, :structure, or a numeric code. :structure is the default. If you are only going to cons lists in your area, you should specify :list so you don't get a useless structure region.

:gc The type of garbage-collection to be employed. The choices are :dynamic (which is the default), :static, and :temporary. :static means that the area will not be copied by the garbage collector, and nothing in the area or pointed to by the area will ever be reclaimed, unless a garbage collection of this area is manually requested. :temporary is like :static, but in addition you are allowed to use si:reset-temporary-area on this area.

:read-only With an argument of t, causes the area to be made read-only. Defaults to nil. If an area is read-only, then any attempt to change anything in it (altering a data object in the area or creating a new object in the area) will signal an error unless sys:%inhibit-read-only (see page 217) is bound to a non-nil value.

:pdl With an argument of t, makes the area suitable for storing regular-pdls of stack-groups. This is a special attribute due to the pdl-buffer hardware. Defaults to nil. Areas for which this is nil may *not* be used to store regular-pdls. Areas for which this is t are relatively slow to access; all references to pages in the area will take page faults to check whether the referenced location is really in the pdl-buffer.

sys:%%region-map-bits
Lets you specify the *map bits* explicitly, overriding the specification from the other keywords. This is for special hacks only.

sys:%%region-space-type
Lets you specify the *space type* explicitly, overriding the specification from the other keywords. This is for special hacks only.

sys:%%region-scavenge-enable
Lets you override the scavenge-enable bit explicitly. This is an internal flag related to the garbage collector. Don't mess with this!

:room With an argument of t, adds this area to the list of areas that are displayed by default by the room function (see page 642).

Example:

```
(make-area ':name 'foo-area
           ':gc ':dynamic
           ':representation ':list)
```

describe-area *area*

area may be the name or the number of an area. Various attributes of the area are printed.

area-list *Variable*

The value of area-list is a list of the names of all existing areas. This list shares storage with the internal area name table, so you should not change it.

%area-number *pointer*

Returns the number of the area to which *pointer* points, or nil if it does not point within any known area. The data-type of *pointer* is ignored.

%region-number *pointer*

Returns the number of the region to which *pointer* points, or nil if it does not point within any known region. The data-type of *pointer* is ignored. (This information is generally not very interesting to users; it is important only inside the system.)

area-name *number*

Given an area number, returns the name. This "function" is actually an array.

si:reset-temporary-area *area-number*

This very dangerous operation marks all the storage in area *area-number* as free and available for re-use. Any data in the area will be lost and pointers to it will become meaningless. In principle, this operation should only be used if you are sure there are no pointers into the area.

If the area was not defined as "temporary", this function gets an error.

See also cons-in-area (page 62), list-in-area (page 65), and room (page 642).

15.2 Interesting Areas

This section lists the names of some of the areas and tells what they are for. Only the ones of the most interest to a user are listed; there are many others.

working-storage-area *Variable*

This is the normal value of default-cons-area. Most working data are consed in this area.

permanent-storage-area *Variable*

This area is to be used for "permanent" data, which will (almost) never become garbage. Unlike working-storage-area, the contents of this area are not continually copied by the garbage collector; it is a static area.

sys:p-n-string *Variable*

Print-names of symbols are stored in this area.

sys:nr-sym *Variable*

This area contains most of the symbols in the Lisp world, except t and nil, which are in a different place for historical reasons.

sys:pkg-area *Variable*

This area contains packages, principally the hash tables with which intern keeps track of symbols.

macro-compiled-program *Variable*

FEFs (compiled functions) are put here by the compiler and by fasload.

sys:property-list-area *Variable*

This area holds the property lists of symbols.

sys:init-list-area *Variable*

sys:fasl-constants-area *Variable*

These two areas contain constants used by compiled programs.

15.3 Errors Pertaining to Areas

sys:area-overflow (error) *Condition*

This is signaled on an attempt to make an area bigger than its declared maximum size.

The condition instance supports the operations :area-name and :area-maximum-size.

sys:region-table-overflow (error) *Condition*

This is signaled if you run out of regions.

sys:virtual-memory-overflow (error) *Condition*

This is signaled if all of virtual memory is part of some region and an attempt is made to allocate a new region. There may be free space left in some regions in other areas, but there is no way to apply it to the area in which storage is to be allocated.

sys:cons-in-fixed-area (error) *Condition*

This is signaled if an attempt is made add a second region to a fixed area. The fixed areas are certain areas, created at system initialization, that are only allowed a single region, because their contents must be contiguous in virtual memory.